Scalability Limitations of VIA-Based Technologies in Supporting MPI

Ron Brightwell and Arthur B. Maccabe

Abstract— **This paper analyzes the scalability limitations of networking technologies based on the Virtual Interface Architecture (VIA) in supporting the runtime environment needed for an implementation of the Message Passing Interface. We present an overview of the important characteristics of VIA and an overview of the runtime system being developed as part of the Computational Plant (Cplant™) project at Sandia National Laboratories. We discuss the characteristics of VIA that prevent implementations based on this system to meet the scalability and performance requirements of Cplant™.**

I. Introduction

Mainstream computer vendors have realized the need for more effective access to networking resources. Studies have proven that, with respect to the actual performance delivered to applications, wide-area network protocols, such as TCP/IP and UDP/IP, do not make effective use of the capability of the underlying network hardware [1]. This performance loss can be attributed to the computational overhead associated with kernel-based transport protocol stacks that enable wide-area networking and the lack of a globally-accepted programming interface that enables efficient overlap of communication and computation. This realization has led to the development of operating system (OS) bypass protocols, such as the Virtual Interface Architecture (VIA) [2], that provide applications with direct access to the network, reducing the amount of interference from the host operating systems in data transfers.

More recently, the development of user-level message passing software technologies has identified performance bottlenecks in the hardware architecture of PC's. The PCI bus has been identified as a source of significant performance penalties in user-level message passing. Several efforts were begun to design a new hardware interface to network components. Among these efforts were Next Generation IO (NGIO) and Future IO (FIO). These efforts have now been consolidated into the InfiniBand Trade Association [3]. While the InfiniBand specification is still under development, it will likely have many characteristics in common with the VIA-based NGIO specification, and most of the performance and scalability limitations of VIA will still be present despite the new hardware interface.

VIA-capable networking hardware is currently available from several vendors: Servernet from Compaq, cLAN from Giganet, and Myrinet from Myricom (when used with a VIA-based Myrinet Control Program). There are also MPI implementations available for these networks: MPI/Pro [4] from MPI Software Technology, Inc., and MVICH [5], a port of MPICH [6] from the National Energy Research Scientific Computing Center at Lawrence Berkeley National Lab. This analysis is not intended to be a general indictment of the VIA specification, VIA-based hardware, or these MPI implementations. Rather, it is intended to address the ability of these technologies to meet specific scalability and performance requirements of the Computational Plant (Cplant™) [7] project.

In the following section, we present a brief introduction to the performance considerations that have motivated "zero-copy" and "OS bypass" protocols. In Section III, we provide background information on the Virtual Interface Architecture [2]. In Section IV we summarize the features of MPI that are important in considering very large scale implementations. In Section V, we describe the Computational Plant machine and the relevant requirements of its runtime system. Section VII presents an analysis of potential difficulties from implementations built on VIA technology in meeting these requirements. Section VIII offers some modifications to the VIA specification that would address some of the scalability and performance limitations. We conclude in Section IX with a summary of the important points of this analysis.

II. Background: Basic Issues in Message Passing Protocols

The improved performance of modern networking technologies has had a significant impact on the development of modern networking protocols. When network bandwidth began to approach the bandwidth of memory copies, protocol implementors and designers quickly looked for ways to eliminate memory copies in message passing protocols. Ultimately, this lead to the development of *zero-copy* message passing protocols. In these protocols, messages are transmitted from application-space to application-space without intermediate store-and-forward copies of the message. The performance cost of any copies incurred between the endpoints of the communication, for example in the network interface card (NIC), is minimized by pipelining the copy. With a zero-copy protocol, the bandwidth observed by an application is the minimum of the network bandwidth and

This work was supported in part by the National Science Foundation CISE Research Infrastructural award CDA-9503064.

R. Brightwell is with the Computational Sciences, Computer Sciences, and Mathematics Center, Sandia National Laboratories, P.O. Box 5800 M.S. 1110, Albuquerque, NM, 87111-1110,(505)845-7432,(505)845-7442 FAX,bright@cs.sandia.gov. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000

A. B. Maccabe is with the Computer Science Department, The University of New Mexico, FEC 313, Albuquerque, NM, 87131, (505)277-6504, (505)277-6927 FAX, maccabe@cs.unm.edu

the bandwidth between the NIC and the memory system.

In designing and implementing zero-copy message passing protocols, the emphasis has been on the elimination of intermediate, system buffers that may be used in the operating system or message passing kernel. Instead of relying on these intermediate buffers, the protocols transmit incoming messages directly into the buffers allocated by the application. In this context, we should note the *zero-cost* copies that have been implemented in a variety of message passing systems including Mach [8]. This method receives incoming messages into system buffers and, instead of copying the messages to application buffers, simply remaps the application's page table entries so that it now uses the pages that were used for the system buffers, instead of its original pages. This approach eliminates the cost of copying message bodies into the application and can be of great benefit in some contexts; however, it places severe alignment restrictions on message reception. The zero-cost copy is not appropriate when these alignment restrictions are not reflected in the application level protocol.

While zero-copy protocols are focused on maintaining end-to-end bandwidth for large messages, they do not address the issues of latency. If the NIC is required to interrupt the operating system for every message reception, this will add a significant latency to the delivery of messages. On the other hand, if the operating system is not involved in the reception of messages, the NIC might violate the protection policies implicit in the operating system. In recent years, a number of protocols have been developed, including VIA [2] and Scheduled Transfer (ST) [9], that incorporate zero-copy and *OS bypass*. These protocols strive to incorporate typical OS protection policies in the message reception code that is run on the NIC, thus *bypassing* the latency that would result from explicitly invoking the operating system on each message reception. Most of the OS bypass protocols use UNIX socket-like, pairwise connections between processes to incorporate the OS protection policies and define the points at which the operating system needs to be consulted (e.g., when establishing a connection, but not when sending or receiving messages using an existing connection).

In addition to end-to-end bandwidth and low latency, processor utilization is another important metric for assessing the effectiveness of a message passing protocol. One relatively easy way to achieve low latency communication is to constantly poll the network looking for incoming messages. As an example, the Intel Paragon has two processors per compute node. Applications that use SUNMOS (Sandia/UNM OS) [10] on this machine can select a mode in which the second processor polls the network or a mode in which the second processor assists in the computation. When the second processor polls the network interface, message passing latencies drop from 27μsecto 17μsec, a decrease of 37%. Having the processor poll the network for incoming messages represents an extreme case of high processor utilization. Many message passing systems require processor cycles in ways that are more subtle, but may prove to be as intrusive. These systems may require that an application-level thread be invoked to accept delivery of large messages, or to manage buffers allocated for "unexpected" messages. While the need for application-level threads may not seem too problematic, these threads complicate the runtime environment needed to support applications. In addition to increasing the total number of threads needed to support applications, these communication threads, by nature, impose significant constraints on the thread scheduling portion of the runtime system. In particular, the thread scheduler must be capable of interrupting a thread that is executing a long computation to ensure that a communication thread is executed in a timely fashion. At best, this will have an adverse impact on computation time; in many cases, it may obviate the advantages of OS bypass.

As part of the Cplant™development effort, we have developed a new message passing system, Portals 3.0 [11], that provides *application bypass* along with zero-copy and OS bypass. Using the Portals 3.0 API, we can implement all of the MPI point-to-point routines while ensuring that the MPI *progress rule* is satisfied without the need for any application-level processing (other than the obvious invocation of the communication routines).

While performance metrics such as high bandwidth, low latency, and low processor utilization are important, predictability in these metrics is also important. If, for example, the latency for transmitting a 512 byte message is usually a few tens of microseconds, but occasionally takes a few milliseconds, application programmers will not be able to tune codes to take advantage of the high performance message passing.

III. VIA

The Virtual Interface Architecture [2], published by Compaq, Intel, and Microsoft, is a specification of the interface between high performance network hardware and computer systems. The goal of this architecture is to reduce the latency associated with message passing by lowering the amount of system software processing needed to move data. VIA gives application processes direct access to the network interface, bypassing the underlying operating system. This strategy eliminates much of the overhead in traditional kernel-based protocol stacks.

A virtual interface (VI) is a point-to-point channel established between two processes. Each end of the channel is composed of a send queue and a receive queue. A process can submit a request, in the form of a descriptor, to either queue to facilitate a data transfer. The requests are processed asynchronously and are updated when the transfer is completed. Processes can then dequeue completed descriptor and re-use it for a subsequent request.

Prior to submitting a request, the memory to be used in a transfer must be locked down. This memory registration process not only insures that the memory to be used is resident, but also allows the network interface to perform virtual address validation and translation.

The VIA specification supports the traditional two-sided send/receive model of communication, as well as a one-

sided remote memory access model.

The send/receive semantics mandate a one-to-one correspondence between send descriptors and receive descriptors. In order for a message to be received, the receiving process must submit a descriptor that corresponds to a descriptor of the sending process. An error will occur if the receive descriptor is not submitted or is not of sufficient size to accept the incoming message. In addition, the specification requires that *both* ends are notified upon completion of the transfer. VIA also supports gather/scatter operations by allowing send and receive operations to specify a set of descriptors to be used in a transfer.

Notification of the completion of a transaction can occur three ways. First, a process can examine the particular queue to which a request was submitted. The process can examine the head of the queue to determine if the request has been completed. A completed request at the head of the queue can then be removed.

In addition, a process can create another queue to poll for requests on a collection of request queues. If a process opens several virtual connections, each with its own send and receive queues, notification of completion of all of these requests can be handled by a single queue. Examining the head of this one queue allows a process to be notified of the completion of requests from several different connections. Once the notification of a completed request is discovered, the request can be dequeued from the queue to which it was originally submitted.

Handlers may also be used to signify the completion of a transaction, although the VIA specification does not fully address the semantics of handlers. A handler may be attached to a send, receive, or completion queue. When the descriptor at the head of the queue is complete, the handler is invoked. The order of handler invocation is unclear. If a receive queue has an associated completion queue, and both queues have associated handlers, the order in which the handlers are invoked upon the completion of a receive descriptor is unspecified.

All queues are traversed in strict FIFO order. Requests that are submitted to a queue are processed, completed, and removed from the queue in FIFO order. Even though requests are processed in order, the actual completion of the transfer may occur out of order. However, the requests must be removed in order regardless of the order of completion.

In addition to the send/receive model, the one-sided model allows a process to specify both the originating buffer and the remote target buffer, provided the remote process has previously set up a region of memory to accept incoming messages. For this type of operation, the target process registers a buffer for remote memory put operations. The target process informs the origin process of the location of the buffer, enabling access to the buffer.

This model also allows for the origin process to specify the origin buffer at the remote process and the target buffer in the local process. This get operation allows the origin process to request that data from the remote process be placed in the memory of the calling process. Get operations require the remote process to notify the local process when the remote buffer has been enabled to accept this type of transfer.

Neither of the get operation nor the put operation consumes a descriptor at the remote process, and the remote process is not notified of any completed transactions. Completion of the one-sided operation at the remote process is inferred through memory inspection or an additional synchronization protocol. Remote memory write operations are a required feature of VIA, while remote memory reads are optional.

IV. MPI

The MPI standard [12] specifies an application programmer interface and semantics for data movement within a set of cooperating processes. It has become the *de facto* standard for user-level message passing in the high performance, scientific computing community.

A key concept in MPI is that of a communicator, which provides a safe message passing context for the multiple layers of software within an application that may need to perform message passing. For example, messages from a support library will not interfere with other messages in the application, provided the support library uses a separate communicator.

Within a communicator, point-to-point operations and collective operations are also independent. An application can post several non-blocking receive operations and then call the MPI barrier routine using the same communicator. Messages used to complete the barrier operation will be processed independently from the posted receive operations. Most implementations of MPI simply use an additional "hidden" collective communicator to distinguish peer communication from collective communication.

In addition to communicators, MPI point-to-point messages also have a an associated user-assigned tag that allows for message selection within a communicator. Tagging individual messages simplifies protocol processing within a communicator.

User-level tags cannot be assigned to MPI collective operations. However, in order for an MPI implementation to allow multiple overlapping collective operations to be in progress, each instantiation of a collective operation requires an additional tag. For example, an application can make successive calls to the MPI broadcast operation using the same communicator. In a spanning tree implementation of the broadcast, the root process need only send the message to a few destinations. The next broadcast operation the root process performs will likely follow the same pattern. However, the destination processes will need to distinguish the first broadcast message from successive broadcast messages in order to preserve MPI's message ordering semantics. An operation counter can be used to tag messages in a collective operation to permit multiple outstanding collective communications to occur.

MPI mandates a fully-connected process model where each process is able to send to any process in the application after the MPI library is initialized. Because an MPI

implementation cannot know the communication patterns of the application *a priori*, point-to-point connections must either be completely established during library initialization, or established as needed. Most MPI implementations establish all connections during initialization to avoid complexity and provide deterministic performance. Establishing connections on a per-use basis requires that a "listener" always be ready to establish a connection, increasing the complexity of the implementation. Low latency operations require that connections be established prior to performing the communication operation.

MPI supports the concept of unexpected messages. While the MPI standard can support completely unbuffered implementations, the protocol overhead incurred by such an approach is usually significant. Low latency for short messages is usually achieved through eager sends with receive-side buffering. The amount of buffering required at the receiver is dependent on several factors, such as network latency and bandwidth, memory copy bandwidth, and communication pattern.

V. Cplant™

The Computational Plant is a large-scale, massively parallel computing resource composed of commodity computing and networking components. The main goal of Cplant™is to construct commodity cluster that is capable of scaling to the order of 10,000 nodes in order to provide the compute cycles required by Sandia's critical applications. Because of this scalability requirement, Cplant™has been designed to address scalability in every aspect of the machine.

The Cplant™runtime system is modeled after the runtime system of the Intel TeraFLOPS [13] machine. This runtime system is dependent upon an underlying high-performance system area network, not only for supporting application message passing with MPI, but also for supporting compute node allocation, application launch, debugging and performance analysis tools, and parallel I/O. The following describes the components of the runtime system.

A. PCT

The Process Control Thread (PCT) runs on each compute node in the cluster and is responsible for controlling the processor and memory resources on the node it controls. The PCT provides the application process on a node with the user's environment as well as the environment needed to participate in a parallel application. It starts the application process, redirects UNIX signals to the application process, attaches the debugger to the application process, terminates the application process, and recovers resources after the application process terminates.

The PCT's communicate with yod and the bebopd (described below) at application launch and throughout the life of a parallel job. A PCT contacts the bebopd to inform the bebopd of the availability of the compute node's resources that the PCT manages. During application launch, the PCT's participating in the parallel job form a group that allows for efficient group communications, such as broadcasts and reductions, using tree-based algorithms. Efficient communication allows the PCT's in a large job to quickly relay global information to compute node processes as well as to the yod process. The latest incarnation of the Cplant™cluster is able to start a 576-node parallel job in less than 10 seconds.

The PCT's implement a space-shared system, where each compute node maps a single parallel application process to each processor on the node. Resources associated with the node–compute, memory, and network–are divided evenly among the application processes. Compute nodes do not support virtual memory paging, and all available physical memory is allocated to the application process.

B. Bebopd

The bebopd is responsible for allocating compute nodes to parallel jobs as requested by user invocations of yod. Bebopd is also responsible for providing compute node status information, such as the number of free compute nodes and which nodes are allocated to which jobs.

Each PCT contacts the bebopd upon startup to make bebopd aware of the available resources. Yod processes contact the bebopd to reserve nodes for the parallel job. The bebopd then contacts the PCT's to insure that each node is ready to participate in the parallel job. The bebopd then passes yod a list of the available compute nodes on which the parallel job will run. Once the PCT's have finished hosting the parallel job, they contact the bebopd to update their availability status.

C. Yod

Yod is the application loader for Cplant™and the main interface to users. Yod contacts the bebopd to allocate a set of nodes, contacts the PCT's to insure their availability, and then communicates with the primary PCT to move the user's environment and executable out on to the compute nodes. Once a job has started, yod serves as an I/O proxy for all UNIX standard I/O functions, including file I/O.

D. Fyod/Sfyod

Fyod and sfyod are yod-like daemon processes that run in an I/O partition to provide a parallel I/O capability. The fyod processes communicate amongst themselves to determine access to secondary storage. The fyod processes also interact with the application processes in the compute partition to perform I/O operations on behalf of the application.

E. Support Tools

Debuggers and performance tools for Cplant™will also rely on the high performance system area network. Debugger processes on each node will likely need the same type of group communication that the PCT's currently use.

F. Computational Steering

The ability to manipulate the data of a running application through a computational steering tool is desirable. It

is imagined that such a tool would need to communicate information to application processes in the compute partition to influence the direction or focus of a computation, or to dynamically manipulate the data set being worked on.

VI. Requirements

The runtime environment and the MPI implementation have requirements that must be met for Cplant™to scale up to ten thousand nodes. Four characteristics of VIA influence the ability to support the requirements of the Cplant™runtime environment and MPI implementation: the number of connections supported, the time needed to establish a connection, the memory requirements for unexpected messages, and performance. The following discusses the scalability effects of these features.

A. Number of Connections

For the runtime system and the MPI implementation, the number of supported connections may be an obvious limitation to scaling. What may not be obvious is the how the number of connections can rapidly increase as the number of computational nodes grows.

Given a cluster with 8192 node, both the MPI implementation and the PCT require fully-connected processes. In addition, each PCT will need a persistent connection to the bebopd for allocation and status messages. Each application process requires a connection back to the hosting yod process for standard I/O.

The allotment of connections for MPI assumes that only a single connection is used per process. This may not be the optimal solution for MPI performance and alternatives are discussed in section VII-A below.

It is also likely that the application process will perform I/O to a parallel file system. In order to stripe data across the parallel file system, each application may have a connection to each I/O node. For a 8192-node cluster, let us assume a ratio of 32 compute nodes per I/O node, resulting in a I/O partition with 256 nodes.

Beyond the basic connections required for simply running an application, connections may also be needed for debugging and communication with processes from external applications.

In order to debug an application running on 8192 nodes, each node will likely run a debugging process. Each debugging process will probably require a connection to a master debugger process. Like the PCT's, these processes will want to establish connections to form a spanning tree to disseminate debugging information in a scalable fashion.

The setup needed for debugging is similar to what is needed to perform computational steering. One can imagine a computational steering tool that has a master process that communicates with the individual application processes to manipulate the process' data while it is running.

A.1 Connections as Needed

The obvious approach to trying to reduce the number of connections required by a large application is to establish connections are they are needed. This approach has several drawbacks:

- Performance degradation
  - Initial send and receive operations may incur the connection establishment cost
  - Initial send and receive operations may incur connection breakdown costs
- Requires a "server" thread waiting to establish incoming connections
  - Consumes CPU cycles for polling the connection
  - Consumes memory for the polling process/thread
- Loss of determinism and predictability
  - Same application can behave markedly different on successive runs
  - Optimized collective routines may have to consider connections as well as network topology
- Loss of fairness
  - Wildcard receives may only come from sources with an established connection
  - Bounded number of connections per interface limits the number of connections across independent processes on a node

B. Time to Open/Close a Connection

For the runtime system, it is crucial that starting a parallel job on thousands of nodes happen in seconds, not tens of minutes or hours. Given a cluster with 8192 compute nodes, each MPI application process will need to establish a connection with every other MPI application process. Let us assume that these connections can be established at every process in O(n) time. In order to establish 8191 connections in a reasonable amount of time, say 30 seconds, each connection will have to be established in

$$\frac{30 \text{ sec}}{8191} = 0.0036 \text{ sec} \cdot \frac{1000 \text{ ms}}{1 \text{ sec}} = 3.6 \text{ ms}$$

While this performance is probably not unattainable, connection establishment is typically a costly operation compared to the data transfer operations. It possibly involves some costly operations, such as system calls at both ends, address resolution, and negotiation protocols. Connection establishment is also typically not a target of intense optimization. While VIA-based hardware vendors are eager to announce latency, bandwidth, and processor overhead performance, none have publicly announced connection establishment timings. The above example also does not include the time needed to establish connections needed for standard I/O to yod or the parallel file system.

Likewise, it is also crucial that ending a parallel job, either through normal or abnormal completion, happen in a reasonable amount of time. Therefore, tearing down a connection needs to be a fast operation, in order to minimize time spent ending a fully-connected parallel application. This overhead will be especially important when terminating an application in order to make resources available, such as when checkpointing or killing a job that has exhausted its allocated resources.

C. Resource Reservation

Given that there are a finite number of connections available, and the total time to establish connections may be significant, reserving connections is a fundamental requirement. A process about to open 8191 connections must be assured that the connections are available from a resource allocation standpoint before the process of establishing connections begins. In order to establish connections as needed, the ability to close a connection and immediately open a new connection is needed. However, there is no guarantee that the process that closes a connection will be able to allocate a new one.

D. Unexpected Messages

For MPI, the amount of buffering for unexpected messages should be a requirement based on the structure of the application's communication pattern and not the number of nodes in the job. Efficient memory usage for message passing is critical when scaling to the magnitude required by Cplant™. In order for memory to be used effectively, the communication subsystem must limit the use of memory to what is absolutely needed, and after having allocated memory, it must make effective use of the memory.

E. Performance

Message passing performance, especially for MPI, is a necessary, but not sufficient, component of scalability. It is the goal of VIA to deliver low latency communication to the application. However, VIA does not provide direct support for the features of MPI that influence the performance of an MPI implementation. For example, MPI requires support for full connectivity, message selection, unexpected messages, and operations on arbitrary regions of application memory. VIA does not support *any* of these features. It is likely that the additional layers of software required to support these features will degrade the achievable performance of the underlying data movement layer.

VII. Analysis

The following is an analysis of specific features of VIA that affect the ability to support the scalability and performance requirements of the Cplant™runtime system and MPI implementation. Many possible implementation strategies exist for both of these communication architectures. The following sections evaluate some of the possible strategies to expose their limitations.

A. Message Selection

VIA does not support message selection within a VI. The memory associated with send and receive descriptors is uniquely identified by a virtual address and a memory handle. There is no support for determining the destination of incoming data based on information supplied in the message. Since MPI has two distinct levels of message selection, one for communicators and one for tags within communicators, this lack of support places the responsibility of MPI message selection within library code running at the user-level. Thus, the host processor must be involved in *all* MPI message passing operations, limiting the amount of overlap of computation and communication that can be achieved.

There are two possible methods for implementing MPI message selection on top of VIA. An MPI process could create a VI for each communicator. For a job containing 64 processes, this method would require each process to have 63 virtual connections established, one for every other process, to support the global communicator MPI_COMM_WORLD. However, since point-to-point operations and collective operations within the same communicator have a separate context, an additional 63 connections would be need to support collective communications on MPI_COMM_WORLD. Thus, for a 64 node job, 126 connections are required to support a single communicator. Using this method for large jobs, it is possible to quickly reach the upper limit on the number of allowable connections. Since this method does not address tag matching on messages within a communicator, user-level processing is still required.

An MPI process could perform all message selection at the user-level. This method is currently used in most MPI implementations where message selection is not supported by the underlying transport layer. Requiring the user-level process to perform message selection mandates that it also perform queue management. Additional system tag, user tag, and MPI protocol information must be sent with each MPI message. As messages arrive, they are inspected to determine their proper location, and the queue of posted receive is searched. If a matching receive is not found, the message is appended to a queue of unexpected messages. If a matching receive is found, the data can be placed at the location the user has specified, either by copying data which accompanied the header or by directly streaming data off of the network. Message selection and queue management at the user-level reduces the amount of processor cycles available to the application, and defeats the purpose of bypassing the host processor.

B. Unexpected Messages

VIA does not have any support for unexpected messages. A receive request must be posted in order for an incoming message to be deposited. Since the MPI standard does not mandate buffering of unexpected messages, it is possible to implement MPI without the additional buffering and protocols necessary to support the arrival of messages for which no corresponding receive has been posted. In practice, the amount of protocol overhead needed to avoid intermediate buffering of messages severely limits performance. Most high-performance MPI implementations implement a two-level protocol to reduce latency for "small" messages and increase bandwidth for "large" messages.

In the short message protocol, the user data accompanies the MPI header. The message is received into a system buffer, the matching criteria is verified, and the user data is copied from the system buffer into the user specified buffer.

In the long protocol, the user data does not accompany the MPI header, and the message is interpreted as a request to send (RTS). Once a matching receive has been posted, the receiver sends a clear to send (CTS) message back to the sender specifying the exact destination of the data. The sending process can then send the user data directly into the user buffer at the receiver. Alternatively, the receiving process could use a remote read operation to get the message from the sender upon receiving the RTS message.

Since the destination process must post a receive in order for a VIA message to be delivered, the MPI implementation must insure that every process always has at least one receive descriptor posted to handle MPI communications. Insuring that a receive descriptor is always posted can be done with either the send/receive operations or the one-sided operations.

When the MPI library is initialized, a process can post a receive to a VI before establishing connection. Once the connection is established, the receive descriptor is activated. The sending process can then send an MPI message that matches this descriptor. The receiver must receive the message, copy the message into an intermediate buffer, post another descriptor, and send a message informing the sending process that it is safe to send another message. This request-to-send(RTS)/clear-to-send(CTS) method of flow control prevents a process from sending messages which are not expected. The number of outstanding sends can be increased so that the sender can generate several messages before waiting for an acknowledgment. Increasing the number of outstanding sends increases the amount of buffer space needed for messages. And, since this is only a single connection, buffer space will be needed for every connection.

The added overhead due to the RTS/CTS protocol is not significant compared to the cost of the message transfer. In effect, the definition of a long message is a message for which the cost of the transfer is significantly larger than the cost of the setup. The need for receive-side management during message reception is based on two considerations: scalability in memory use and the additional complexity needed to manage the receiver's message space.

First, we consider the issue of scalability in memory use. Suppose that the short message size is 4096 bytes, we have a machine with 4096 nodes, and each sender could have 4 outstanding short messages. Without receiver-management during the receive operation, the receiver will need to reserve space for all of the messages that the individual senders could send. This results in a total of:

$$4096\text{nodes} \cdot 4096\text{Bytes} \cdot 4\text{messages} = 64\text{Mbytes}$$

Second, an additional protocol needs to be implemented in order to know when it is safe to re-use slots. Even if the underlying network has flow control at the network layer, another level of flow control is needed for the MPI library to insure that consecutive send operations do not overwrite previous send operations that have not been processed by the receiver.

One sided operations could also be used to insure that a receive descriptor is always posted. The above strategy RTS/CTS strategy could be implemented using remote memory access operations. Each process could open up an area of memory into which the sending process deposits MPI messages. This method would eliminate the need for the receiving process to continually post receives. However, since there is no notification of a completed put operations at the target, the receiving process will have to loop through the set of buffers looking for changes in memory to determine message arrival. However, this method of searching for messages is not scalable, since the time needed to perform the loop will increase as the number of processes increases. And, using one-sided communications does not eliminate the need for the RTS/CTS flow control.

C. Arbitrary Message Buffers

The guidelines that VIA offers suggest that processes should not register and unregister memory regions frequently, as this can lead to fragmentation of the page tables on the network interface. MPI has no restriction on the location of message buffers or on the number of buffers that can be posted. Since VIA provides no mechanism to determine whether a region of memory has been registered, long protocol send and receive buffers will have to be registered when initiated and unregistered when completed. The process could potentially impact applications which transfer large amounts of data. Unfortunately, ping-pong latency tests do not include the time needed to register memory.

D. Completion Semantics

The RTS/CTS protocol needed to handle unexpected messages could be implemented using a single VI. However, due to the strict FIFO ordering of posted requests, it may be sensible to use a VI for short protocol, RTS, CTS, and acknowledgment messages, and a separate VI for long protocol data messages. The strict ordering of FIFO requests does not allow an MPI implementation to process requests in the order they complete. MPI has support for waiting for the completion of a group of requests. While the pairwise order of messages is preserved, MPI allows an implementation to process messages in the order they have completed. The VIA semantics do not allow an MPI implementation to process shorter messages while waiting for the completion of a larger message.

The completion semantics of VIA send operations are also tighter than what MPI mandates. Completion of a VIA send operation implies that the receive operation has also completed. Whereas in MPI, completion of a standard mode send operation only implies that the send buffer is available to application. The non-local completion semantics of VIA are also more restrictive than the MPI synchronous mode send, which can complete when a matching receive operation has *started* at the destination.

VIII. Extensions to VIA

Some of the limitations of VIA that we have presented could possibly be overcome by extending the semantics outlined in the current specification.

The concept of a connection bundle has been widely used as an abstraction for aggregating endpoint resources [14]. Bundles allow an application to treat and manipulate connections as a single resource. For example, a single API call would be needed to create multiple connections, thus allowing the underlying implementation to know how many connections are required and possibly provide an optimized path for allocating them. Connections in a bundle could be replaced individually, in order to conserve resources when closing and re-opening a connection.

Rather than assigning specific receive buffers to each VI, a single pool of buffers could be shared between multiple VI's. A VI to which no receive descriptor has been posted would simply allocate a buffer out of the pool of buffers to process an incoming message. This mechanism would reduce the amount of buffering required for unexpected messages on large numbers of VI's.

One solution to the problem of connection establishment times is to implement VIA on top of a connectionless transport layer. This strategy would essentially eliminate connection operations. For example, a VIA implementation on top of the Portals 3.0 API on Myrinet in the Cplant™environment would not incur any connection establishment or breakdown overhead, since processes are fully connected when they are created.

IX. Summary

In this paper we have presented an analysis of the ability of VIA to support the Cplant™runtime environment as well as a high performance implementation of MPI. We have provided an overview of VIA, the components of the Cplant™runtime system, and have identified the specific characteristics of VIA that may impact scalability and performance. Some of these characteristics may carry over from the VIA-based communication platforms, such as Giganet and Servernet, to the InfiniBand technology.

References

[1] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler, "High-Performance Local Area Communication With Fast Sockets," in *Proceedings of USENIX'97*, 1997.

[2] Compaq, Microsoft, and Intel, "Virtual Interface Architecture Specification Version 1.0," Tech. Rep., Compaq, Microsoft, and Intel, December 1997.

[3] Infiniband Trade Association, *http://www.infinibandta.org*, 1999.

[4] Rossen Dimitrov and Anthony Skjellum, "An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing," in *Proceedings of the Third MPI Developer's and User's Conference*, March 1999, pp. 15–24.

[5] National Energy Research Scientific Computing Center, *MVICH - MPI for Virtual Interface Architecture*, 1999, http://www.nersc.gov/research/ftg/mvich/index.html.

[6] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.

[7] R. B. Brightwell, L. A. Fisk, D. S. Greenberg, T. B. Hudson, M. J. Levenhagen, A. B. Maccabe, and R. E. Riesen, "Massively Parallel Computing Using Commodity Components," *Parallel Computing*, vol. 26, no. 2-3, pp. 243–266, February 2000.

[8] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A new kernel foundation for UNIX development," in *USENIX Conference Proceedings*, Atlanta, GA, 1986, USENIX.

[9] Task Group of Technical Committee T11, "Information Technology - Scheduled Transfer Protocol - Working Draft 2.0," Tech. Rep., Accredited Standards Committee NCITS, July 1998.

[10] Arthur B. Maccabe, Kevin S. McCurley, Rolf Riesen, and Stephen R. Wheat, "SUNMOS for the Intel Paragon: A brief user's guide," in *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, June 1994, pp. 245–251.

[11] R. B. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen, "The Portals 3.0 Message Passing Interface," Tech. Rep. SAND99-2959, Sandia National Laboratories, December 1999.

[12] Message Passing Interface Forum, "MPI: A Message-Passing Interface standard," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 8, 1994.

[13] Sandia National Laboratories, *ASCI Red*, 1996, http://www.sandia.gov/ASCI/TFLOP/ Home_Page.html.

[14] Alan Mainwaring and David Culler, "Active Message Applications Programming Interface and Communication Subsystem Organization," Tech. Rep., Computer Science Division, University of California at Berkeley.